INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Modeling Railway Control Systems using Graph Grammars: a Case Study

Anne-Alexandra Holzbacher, Michaël Périn, Mario Südholt

N° 3210

juillet 1997

THÈME 2

ISSN 0249-6399

INRIA
RENNES

Modeling Railway Control Systems using Graph Grammars: a Case Study

Anne-Alexandra Holzbacher*, Michaël Périn†, Mario Südholt†

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n ° 3210 — juillet 1997 — 24 pages

Abstract: In this report, we develop a railway control system. We are mainly concerned with the *software architecture* of the control system and its *dynamic evolution*; we do not discuss here the implementation details of the components forming the control system. The software architecture is characterized by a hierarchy of controllers whose leaves are local controllers connected in a network that mimics the underlying railway topology. Using a particular *graph grammar*, we formally define a class of software architectures for the railway control system ensuring several desirable properties by construction. The dynamic evolution of the architecture is modelled by a set of *coordination rules* which define graph transformations. Particular emphasis is placed on the *verification* of these rules with respect to desirable properties encoded in the grammars. Using the graph grammar and the coordination rules as a formal specification of the railway control system, we derive an implementation in ConCoord, an environment for concurrent coordinated programming whose coordination language permits us to define the software architecture of the control system and its dynamic evolution abstracting away from the implementation details of its components.

Key-words: software architecture, graph grammar, architecture style, dynamic architectural modification, formal methods

(Résumé : tsvp)

This report is an extended version of the article presented at COORDINATION '97 [HPS97].

*aholzbac@insa-rennes.fr, IRISA/INSA Rennes, Département d'Informatique
†{mperin,sudholt}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 02 99 84 71 00 – Télécopie : (33) 02 99 84 71 71

Modélisation d'un système de contrôle ferroviaire à l'aide de grammaires de graphes

Résumé : Ce rapport présente la conception d'un système de contrôle pour réseaux ferrés. Nous nous sommes principalement attachés à décrire en termes de graphes et de transformations de graphes, *l'architecture logicielle* du système de contrôle et *l'évolution de cette architecture* ; nous ne décrivons pas en détail l'implémentation des composants du système. L'architecture logicielle que nous proposons pour le système de contrôle est hiérarchique. Il s'agit d'un arbre de contrôleurs dont les feuilles sont des contrôleurs locaux connectés en réseau de manière à reproduire la topologie du réseau ferré. À partir de *grammaires de graphes*, nous définissons formellement une classe d'architectures qui vérifie par construction des propriétés structurelles importantes pour le système de contrôle. L'architecture du système est susceptible d'évoluer en cours d'exécution, entre autres pour prendre en compte les modifications topologiques du réseau ferré. Les modifications dynamiques de l'architecture sont modélisées par des *règles de coordination* qui définissent des transformations sur les graphes des architectures. Nous sommes en mesure de vérifier que les règles de coordination préservent les propriétés structurelles induites par les grammaires. Outre leur utilisation à des fins d'analyse, les règles de coordination servent de spécifications formelles pour construire un prototype du système de contrôle. Nous utilisons ConCoord, un environnement pour la programmation concurrente coordonnée, dont le langage de coordination, CCL, permet de décrire l'architecture logicielle du système et son évolution dynamique en faisant abstraction des détails d'implémentation des composants. Les similitudes entre les règles de coordination et leur implémentation en CCL nous permettent d'envisager une automatisation partielle du processus d'implémentation.

Mots-clé : architectures logicielles, grammaires de graphes, style architectural, modification dynamique d'architectures, méthodes formelles.

Contents

1 Introduction

In this paper, we propose a design for a railway control system following the requirements of [dJ97]. The controlled railway network consists of tracks, junctions and stations with one or more platforms. Each train traverses the network following a global schedule which defines a timetable and a route consisting of stations. The railway control system communicates with trains via a mobile wide area network (MWAN). It must ensure that trains do not collide and that they respect as much as possible their schedule. It may make travelling more comfortable by avoiding abrupt changes in train speeds. In addition, the control system must provide the state monitoring of the trains on the railway network and execute corrective actions on train schedules if needed. The topology of the railway network may change for example via the addition of tracks. Trains may be dynamically introduced or removed at stations of the railway network. Generally speaking, most parameters of the railway network can vary during the execution of the control system.

We have added one assumption to the problem definition: we require for each train a detailed schedule which defines its timetable and its route giving *all* the places that it must traverse, *i.e.* tracks, junctions and platforms. This schedule is defined off-line. During program execution, a train schedule may be subject to corrective actions by the control system. With regard to the requirements defined in the case study [dJ97], the solution we present addresses at a high-level of abstraction most requirements except for the fault-tolerance aspects. In the software architecture we propose next, the functionalities of the railway control system are distributed among its components in a manner that promotes real-time responsiveness. We do not discuss here the implementation details of the system components. It is worth noticing that in a programming environment with a separate definition of the system architecture and the component codes (*e.g.* in ConCoord), each component code can be expressed in the programming language which better suits its functionality, for instance real-time aspects.

The paper is structured as follows. First, we informally discuss our design proposal for the railway control system (Section 2). Second, we formally define the style of the software architecture of the control system using graph grammars and we describe the dynamic evolution of the architecture in terms of coordination rules (Section 3). Third, we implement a coordination rule in ConCoord [Hol96] (Section 4). Finally, we discuss the limitations of our approach in terms of ease of use and expressiveness.

2 An Informal Design of the Railway Control System

Complex system can be seen as a set of individual components and an architecture defining their links and interactions. An explicit description of a system architecture provides a global high-level view of the system which facilitates the different phases of software engineering [Gar95, SG95]. Here, we present the first design phase of a railway control system: the informal definition of its components and its software architecture. We propose a solution in which control of the railway is first distributed among local controllers connected in a network and then centralised using a control hierarchy (see Figure 1).

Local Controllers. We define a local controller for each track, junction and platform of the railway network (*cf.* Figure 1 right). A local controller for a railway device communicates in real-time with the trains currently at the device and may modify their detailed schedule.

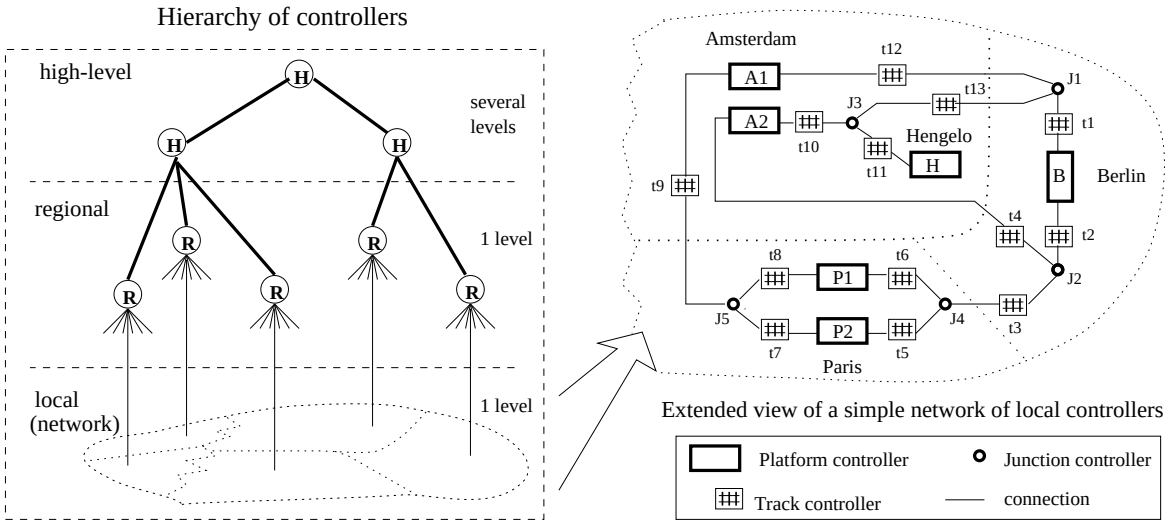


Figure 1: Architecture of the Railway Control System

A track controller may modify the speed of trains currently on the track; a junction controller may redirect trains through the junction to resolve potential collisions; a platform controller may delay trains and handles the addition and removal of trains in the railway network. We do not provide local controllers for stations; a train at a station is managed by the controller of the station platform where it is. Whenever possible, schedule constraints are solved by local controllers in order to promote real-time responsiveness. This distribution of control between local controllers provides means for concurrency and thus naturally leads to a distributed execution of local controllers on a computer network.

Network of Local Controllers. Local controllers have a view of the system state which is limited to the trains currently at the device they control. In order to decide corrective actions on schedules, a local controller may require information about the trains located at neighbouring devices. Thus, it may interact with the controllers managing its neighbouring devices. A track controller communicates with two junction controllers, two platform controllers or a junction and a platform controller. A junction controller interacts with three track controllers. For simplicity, we only consider junctions linking three tracks which can serve to model junctions linking more than three tracks. A platform controller communicates with one or two track controllers. The interactions between local controllers define a software architecture which mimics the topology of the railway network.

Control Hierarchy. Due to their limited view of the system state, local controllers are not able to resolve all constraints on the trains schedules. Moreover, the monitoring of the system also requires a more centralised view of the system state. For these reasons, we superimpose a hierarchy of regional and high-level controllers over the network of local controllers (*cf.* Figure 1 left). First, we partition the railway network into regions connected by tracks. A regional controller centralises state information and controls decisions corresponding to a region of the railway network. It interacts will all local controllers of the region, requesting their state, getting their alarms and sending them commands with corrective actions. Each

local controller in the railway control system communicates with a single regional controller. Second, we develop on top of the regional controllers a hierarchy of high-level controllers in which controllers at consecutive levels interact in the same way that local and regional controllers do. If a local controller cannot resolve a constraint, this is propagated upwards in the control hierarchy until a controller which can handle it is found. For example, a high-level controller may solve schedules constraints for trains crossing a region frontier.

The monitoring of the system state is realised by the regional and high-level controllers. The users interact with the railway control system through these controllers, in order to request, for example, a modification on the topology of the railway network, such as the addition of a platform on an existing track. The number of levels of the hierarchy is critical for the system performance. Its value at system start-up depends on the initially foreseen load for each control region of the railway network. During system execution, load balancing can be done by dynamically adding/removing controllers to/from the hierarchy.

3 Formal Definition using Graph Grammars

The design suggested in the previous section is too informal to provide a means for the verification of an implementation of the railway control system; a formal definition of its software architecture is needed. Moreover, a formal definition of the software architecture may be used to derive an implementation of the railway control system. In Section 3.1, we introduce a formal framework for the definition of software architectures based on graph grammars. We generalise it to a notion of *multiple views* of a software architecture. Using this framework, we define in Section 3.2 the network of local controllers, the control hierarchy of the railway control system and the relation of these two views. Finally, the dynamic evolution of the software architecture is investigated in Section 3.3.

3.1 Describing Software Architectures Using Graph Grammars

The formal framework proposed by D. Le Métayer in [Mét96] supports the description of classes of architectures usually called *architectural styles* [AAG95]. The notion of style is formally defined in this framework as a set of architectures that have similar structure defined using graph grammars.

Graph Representation. Software architectures are commonly defined as graphs in order to formalise the "box and line" drawings frequently used in informal descriptions. Nodes of such a graph stand for system components and edges represent interactions between components. We represent graphs formally as multisets of relations $R(c_1, \ldots, c_n)$ where R is an n-ary relation over component names $c_1, \ldots, c_n$. We distinguish two kinds of relation:

- a relation whose name begins with a capitalized letter, such as $\mathbf{Process}(c_1)$, specifies that the type of the component c_1 is **Process**. Such relations correspond to nodes of the architecture graph.
- relations named using only lower-case letters, such as $\mathbf{pipe}(c_1, c_2)$, denote links between several components. In terms of the graph representation, such a relation represents an edge labeled by **pipe** between the nodes c_1 and c_2.

In the sequel, the words "multiset" or "graph" are treated as synonymous because they are two formal representations of the same concept. The representation of a graph as a multiset enables us to generate graphs through context-free graph grammars and define graph transformations as multiset rewrite rules.

3.1.1 Graph Grammars.

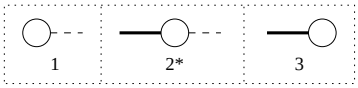
We now introduce graph grammars as a means of the constructive definition of sets of correct architectures. A graph grammar is a finite representation of a (possibly) infinite collection of finite graphs. A graph grammar is defined as usual as a four tuple $\langle NT, T, PR, AX\rangle$, where NT and T are the sets of non-terminals and terminals, respectively, PR is a set of production rules and AX is a special non-terminal, the axiom, that initiates the production process.

Graph grammars generate terms containing variables denoting component names. The set of terminals T (resp. NT) is the set of terms built from relation names drawn from a finite set Tn (resp. NTn) and such variables.[1] Terminals (written in bold face) denote basic blocks of the architecture and non-terminals (written in italic) can be interpreted as constructors of parts of an architecture. We represent a grammar by its set of production rules where the axiom is marked with the symbol '$\triangleright$'. Terminals and non-terminals are obvious from the production rules.

Since we consider context-free grammars with variables, production rules are of the form $lhs \rightarrow rhs$ where lhs is a *single* non-terminal $A(x_1, \ldots, x_i)$ and rhs is a multiset of terms $B(y_1, \ldots, y_j) \in NT \cup T$. The production rules form a multiset rewrite system. Given a multiset M, the application of a rule consumes the lhs term from M and adds the rhs terms to M, thus yielding to a new multiset. In a production rule, all variables that only appear on the rhs receive a fresh name, that is, a name not already used in the multiset. We are exclusively interested in the *terminal* multisets produced by a grammar G, that is, the multisets containing only terminal terms, because the terminals terms form the graph of the architecture.

Consider, for instance, the following grammar, which defines the style of pipeline architectures:

$$
\begin{array}{lll}
\triangleright\ \mathit{Pipeline} & \rightarrow_1 & \mathbf{Process}(c_1),\ \mathit{Pipe}(c_1) \\
\mathit{Pipe}(c_1) & \rightarrow_2 & \mathbf{pipe}(c_1, c_2),\ \mathbf{Process}(c_2),\ \mathit{Pipe}(c_2) \\
\mathit{Pipe}(c_1) & \rightarrow_3 & \mathbf{pipe}(c_1, c_2),\ \mathbf{Process}(c_2)
\end{array}
$$



graphical representation of the pipeline architecture style.

$\mathbf{Process}(c_1)$ and $\mathbf{pipe}(c_1, c_2)$ represent a process component and a pipe between two components, respectively. An instance of a pipeline architecture is derived from the *Pipeline* axiom. Rule 1 introduces a process component with a fresh name. Rule 2 is used to add new pipeline connections and process components. Finally, Rule 3 adds the last connected process component.

3.1.2 Architecture Style and Multiple Views

A grammar is a finite representation of the infinite set of terminal graphs that it can produce. A software architecture belongs to the style defined by a grammar if its corresponding graph

[1] More complete and precise introduction to graph grammars can be found in [Cou90].

can be produced by the grammar. Thus, the architecture style defined by a grammar G can be defined as (see [Mét96]):

$$ArchStyle(G) = \{M \mid \{Axiom_G\} \rightarrow_G^* M \text{ and } M \text{ is terminal}\}$$

We extend this definition to include multiple architectural views.[2] Each view, say V_i, has its own style defined by a grammar G_i and is related to the set T_i of terminals of G_i. The overall style is defined by a tuple of grammars $(G_1, \ldots, G_n)$ as follows. An architecture represented as a multiset M of terminals in $T_1 \cup \ldots \cup T_n$ belongs to $ArchStyle((G_1, \ldots, G_n))$ if, for each view V_i, M restricted to T_i belongs to the style defined by G_i.

$$ArchStyle((G_1, \ldots, G_n)) = \{M \mid \bigwedge_{i=1,\ldots,n} (\, M \cap T_i \in ArchStyle(G_i)\,)\}$$

3.1.3 Coordination Rules

The dynamic evolution of software architectures is described in terms of *multiset rewrite rules*, henceforth called *coordination rules*, and denoted by $lhs \Rightarrow rhs$. Their semantics is the same as that for rewrite rules of context-free graph grammars but, in contrast, there are no restrictions on the *lhs* number of terms. The rule shown below, for example, reduces a pipeline by discarding a process and replacing the two pipes connected to it by a new pipe.

$$\mathbf{pipe}(c_1, c_2),\ \mathbf{Process}(c_2),\ \mathbf{pipe}(c_2, c_3) \quad \Rightarrow \quad \mathbf{pipe}(c_1, c_3)$$

An application of this rule to a pipeline architecture always yields a pipeline and thus preserves the pipeline style. This notion of the preservation of architecture style is at the core of our approach and can be checked *statically* by verifying the correctness of a coordination rule with respect to a grammar G using the algorithm defined in [FM96].

In the process of the formal definition of architectures, this algorithm can also be used to check *statically* that a given architecture M belongs to the style defined by a grammar. The algorithm has been implemented and is currently under testing. It is restricted to single view architectures but its extension to multiple views is straightforward: coordination rules must be checked with respect to each grammar related to a view of the multiple view architecture style.

3.2 Railway Topology View and Control View

As discussed in Section 2, the software architecture of the railway control system consists of two related structures: a network of local controllers and a control hierarchy (*cf.* Figure 1). In this section, we define theses structures by means of graph grammars.

Notational Extensions. The formal definitions are expressed using standard notations from the field of rewrite systems. However, we use three minor notational extensions in order to enhance the readability of grammars and coordination rules.

- $R\{(x_1, x_2), (y_1, y_2)\}$ is a syntactic sugar for $R(x_1, x_2),\ R(y_1, y_2)$

[2] In this case study, we limit our description to 2 views: the local network of controllers and the hierarchy of controllers.

- $[\![C]\!]: lhs \Rightarrow rhs$ is equivalent to $lhs \cup C \Rightarrow rhs \cup C$. Intuitively, C defines the context in which the coordination rule is applied.
- $n \times R(x)$ is equivalent to n occurrences of the same term $R(x)$.

3.2.1 Definition of the Network of Local Controllers

Graph Representation. As suggested in Section 2, we have three kinds of local controllers, one per type of railway device (that is, track, platform and junction). In terms of multisets, these controllers are represented using unary relations and their connections using a binary relation:

- $\mathbf{Platform}_n(p)$ represents a platform controller with n connecting ports, $n \in \{1, 2\}$.
- $\mathbf{Junction}(j)$ represents a junction controller with three connecting ports.
- $\mathbf{Track}(t)$ represents a track controller with two connecting ports.
- $\mathbf{connection}(t, e)$ represents a connection between a port of the track controller t and a port of the local controller e (where e is a platform or a junction controller).

These relations place conditions on the connectivity of local controllers: the cardinality of the connections for each type of local controller and the type of the connections $\mathbf{connection}(t, e)$ must be respected. In particular, a track controller has to have always two connections which can only be of type $\mathbf{connection}(t, e)$. The respect of these constraints lies on the grammar that describes the correct networks of local controllers.

Graph Grammar G_1 and its Properties. In Figure 2, we define the grammar G_1 for the network of local controllers and represent graphically five of its rules. Rule 1 expresses the fact that a general control network can be made of several disconnected subnets. Rules 3 and 4 define the smallest control networks as consisting of two platform controllers and a path between them, or a platform controller fully connected by a circuit. Rule 7 provides a way of adding a dead end platform controller to a network using a junction. Rule 8 introduces a split/merge of a path using two junctions and corresponding paths. This last scheme is a basic transformation used to produce more complex networks.

Examples of networks that belong to the style of the railway topology grammar are the network in Figure 1 right defined in the presentation of the case study [dJ97] and an interchanger network using a four way junction. Their derivations are shown in Appendix A.

Grammar G_1 ensures that all networks belonging to its style obey two *connectivity properties* by construction. First, it respects the cardinality of connections for each kind of local controller, and thus can not produce a local controller which is disconnected from all other local controllers. For example, a track is always connected to *exactly* two controllers because a terminal **Track** can be produced only by applying Rule 5, which connects the track to two other controllers at the same time.[3] Second, the grammar guarantees that the network is well-formed in the following sense: it cannot produce junctions and platforms which are not directly connected. There must be at least an intermediate track. These properties have to be proved by induction on the production rules, we will discuss that point in Section 5.

[3]The semantics of graph grammars requires to create fresh names for all variables that only appear on the right hand side of a rule. As a consequence it ensures that the identifier of the track controller introduced in Rule 5 would never be reused elsewhere in order to create incorrect new connections.

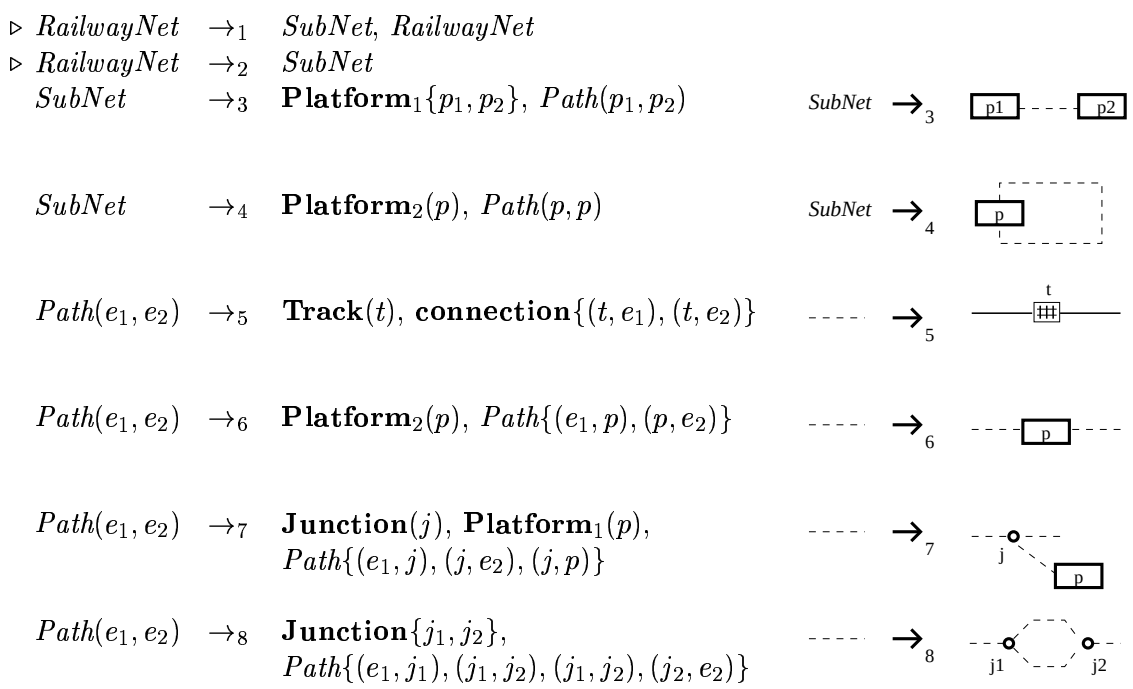


Figure 2: Grammar G_1 for the Railway Topology

3.2.2 Definition of the Control Hierarchy

Local controllers are not sufficient to satisfy all requirements of the case study. They are not able, for instance, to solve all the constraints on train schedules. For this reason, we superimpose a hierarchy of regional controllers (henceforth R-controllers) and high-level controllers (HL-controllers) on the network of local controllers (L-controllers) (*cf.* Figure 1 left).

Graph Representation. The three levels of the control hierarchy are clearly separated by using different terminals representing controllers located at different levels.

- The term **R-Ctrler**(c) denotes a R-controller which supervises a region in the railway network. It can control at most max' L-controllers.
- **ctrl**(c, l) denotes a link between a R-controller c and a L-controller l.
- **HL-Ctrler**(c) represents a HL-controller supervising R-controllers or other HL-controllers. It may control at most max sub-controllers.
- **supervise**(c, c') represents a link between a HL-controller c and a HL-controller or R-controller c'.
- **HL-freelink**(c) (resp. **R-freelink**(c)) represents free control links of an HL-controller (resp. R-controller) and models the remaining capacity.

Graph Grammar G_2 and its Properties. The grammar shown in Figure 3 defines the architecture of the control hierarchy of the railway control system. Rules 1-5 build the hierarchy of high-level controllers as a tree whose leaves, created by Rule 2, are R-controllers. Each HL-controller is created by Rule 3 with *max* links (both **HL-freelink** and **supervise** links), where *max* is a constant that must be set as part of the definition of the grammar. The free links allow us to know how many controllers a HL-controller manages. This information is used in several coordination rules defined below to test if an HL-controller supervises anything. We could have attached to each HL-controller the list of subcontrollers it manages using the list representation proposed in [FM96] in order to represent the same information. This would have allowed a HL-controller to supervise an unbounded number of lower-level controllers. The assumption that HL-controllers only have a limited capacity seems to be more realistic, however. Rules 6–8 define the connection between R-controllers and L-controllers. Here, free links are used to limit the number of L-controllers of an R-controller to max'. For an R-controller, free links are denoted by **R-freelink**. Rules 9–12 list the type of local controllers which can be linked to a R-controller

By construction, Grammar G_2 ensures three properties: first, the simplest control hierarchy consists of one R-controller. Second, an R-controller must have exactly one HL-controller, that is, in a correct system, R-controllers are connected to the global control. Third, the grammar does not produce R-controllers or HL-controllers which control nothing, since both **HL-Ctrler** and **R-Ctrler** control at least one subcontroller (Rule 3 and Rule 6).

3.2.3 Relating the two Views

Grammars G_1 and G_2 can be interpreted as defining two different and complementary architectural views of the railway control system. The first represents the neighbour relationship between the network devices managed by the L-controllers; the second enables the distribution/centralization of control by means of a control hierarchy. The coherence of both views is ensured by the definition of multiple architectural views in Section 3.1.

Let T_1 (resp. T_2) be the set of terminals of G_1 (resp. G_2) and *RCS* the multiset representing an architecture. *RCS* contains terminals of both T_1 and T_2. The railway topology view and the control hierarchy view can be formally defined in terms of the multiset *RCS*: the view which corresponds to grammar G_1 (G_2) is the multiset *RCS* filtered with respect to the terminals of G_1 (G_2), that is $RCS \cap T_1$ ($RCS \cap T_2$). For example, if we are only concerned with the local controllers defined by Grammar G_1, we filter the multiset *RCS* with respect to the set of terminals T_1, effectively hiding terminals of Grammar G_2 (**HL-Ctrler**, **supervise**, **R-Ctrler**, **ctrl**, **HL-freelink**).

Assuming that *RCS* belongs to $ArchStyle((G_1, G_2))$ then all terms of *RCS* generated by both G_1 and G_2 are the same in the two views and the two views can be "glued" on their commons terms. In our case, common terms are terminals representing L-controllers (that is, names of relations in $T_1 \cap T_2$). Therefore the coherence of the two related views and the property of Grammar G_2 ensures that each L-controller is linked to exactly one R-controller.

3.3 Definition of Dynamic Architectural Changes

In this subsection, we define a set of coordination rules governing dynamic changes of the network of L-controllers and the control hierarchy. Correctness of these rules with respect to the two grammars ensures preservation of the style defined by (G_1, G_2). Therefore properties

a) High-level controllers (max is an integer constant)

▷ *Tree* $\rightarrow_1$ *SubTree*(c)

SubTree(c) $\rightarrow_2$ *R-Ctrl*(c)

SubTree(c) $\rightarrow_3$ **HL-Ctrler**(c), **supervise**(c, c'), *SubTree*(c'), $(max-1)\times$ *HL-Link*(c)

HL-Link(c) $\rightarrow_4$ **supervise**(c, c'), *SubTree*(c')

HL-Link(c) $\rightarrow_5$ **HL-freelink**(c)

b) Regional controllers (max' is an integer constant)

R-Ctrl(c) $\rightarrow_6$ **R-Ctrler**(c), **ctrl**(c, l), *L-Ctrl*(l), $(max'-1)\times$ *R-Link*(c)

R-Link(c) $\rightarrow_7$ **ctrl**(c, l), *L-Ctrl*(l)

R-Link(c) $\rightarrow_8$ **R-freelink**(c)

L-Ctrl(l) $\rightarrow_9$ **Platform**$_1$(l)

L-Ctrl(l) $\rightarrow_{10}$ **Platform**$_2$(l)

L-Ctrl(l) $\rightarrow_{11}$ **Junction**(l)

L-Ctrl(l) $\rightarrow_{12}$ **Track**(l)

Figure 3: Grammar G_2 for the Hierarchic Control Structure

which rely on the style still hold after dynamic modifications through correct coordination rules.

3.3.1 Changing the Railway Topology Dynamically

In Figure 4 we show the coordination rules governing the dynamic changes to the network of L-controllers (*i.e.* insertion/elimination of controllers) which mimic transformations on the devices of the railway topology. In order to make coordination rules more readable, we will denote the linking of a track controller t to two controllers e_1 and e_2 by **Track**($t\langle e_1, e_2\rangle$)

a) correct introduction rules

$$\mathbf{Track}(t\langle e_1, e_2\rangle) \quad \Rightarrow_1 \quad \mathbf{Platform}_2(p),\ \mathbf{Track}\{t_1\langle e_1, p\rangle, t_2\langle p, e_2\rangle\}$$

$$\mathbf{Track}(t\langle e_1, e_2\rangle) \quad \Rightarrow_2 \quad \mathbf{Junction}\{j_1, j_2\},\ \mathbf{Track}\{t_1\langle e_1, j_1\rangle, t_1'\langle j_1, j_2\rangle, t_2'\langle j_1, j_2\rangle, t_2\langle j_2, e_2\rangle\}$$

$$\mathbf{Platform}_1(p) \quad \Rightarrow_3 \quad \mathbf{Platform}_2(p),\ \mathbf{Track}(t\langle p, p'\rangle),\ \mathbf{Platform}_1(p')$$

$$[\![\mathbf{Platform}_1\{p_1, p_2, p_3, p_4\}]\!]:\ \mathbf{Track}\{t\langle p_1, p_2\rangle, t'\langle p_3, p_4\rangle\} \quad \Rightarrow_4 \quad \mathbf{Junction}\{j_1, j_2\},\ \mathbf{Track}(t''\langle j_1, j_2\rangle),\ \mathbf{Track}\{t_1\langle p_1, j_1\rangle, t_2\langle j_1, p_2\rangle, t_1'\langle p_3, j_2\rangle, t_2'\langle j_2, p_4\rangle\}$$

b) correct elimination rules

$$\mathbf{Track}\{t_1\langle e_1, j_1\rangle, t_2\langle j_2, e_2\rangle, t_1'\langle e_3, j_1\rangle, t_2'\langle j_2, e_4\rangle\},\ \mathbf{Junction}\{j_1, j_2\},\ \mathbf{Track}(t''\langle j_1, j_2\rangle) \quad \Rightarrow_5 \quad \mathbf{Track}\{t\langle e_1, e_2\rangle, t'\langle e_3, e_4\rangle\}$$

$$[\![\mathbf{Platform}_2\{p_1, p_2\},\ \mathbf{Track}(t'\langle p_3, e\rangle)]\!]:\ \mathbf{Platform}_2(p_3),\ \mathbf{Junction}(j),\ \mathbf{Track}\{t_1\langle p_1, j\rangle, t_2\langle j, p_2\rangle, t''\langle j, p_3\rangle\} \quad \Rightarrow_6 \quad \mathbf{Platform}_1(p_3),\ \mathbf{Track}(t\langle p_1, p_2\rangle)$$

c) two incorrect rules

$$\mathbf{Track}(t\langle e_1, e_2\rangle) \quad \Rightarrow_7 \quad \epsilon$$

$$\mathbf{Track}\{t\langle p_1, p_2\rangle, t'\langle p_3, p_4\rangle\} \quad \Rightarrow_8 \quad \mathbf{Junction}\{j_1, j_2\},\ \mathbf{Track}(t''\langle j_1, j_2\rangle),\ \mathbf{Track}\{t_1\langle p_1, j_1\rangle, t_2\langle j_1, p_2\rangle, t_1'\langle p_3, j_2\rangle, t_2'\langle j_2, p_4\rangle\}$$

Figure 4: Coordination Rules for the Network of Local Controllers

instead of the notation $\mathbf{Track}(t)$, $\mathbf{connection}(t, e_1)$, $\mathbf{connection}(t, e_2)$. Similar simplifications will be applied in the graphics in order to make them more intuitive: we shall no more explicitly represent track controllers. Expressed in graphical form, the coordination rules are quite easy to understand. Figure 5 left shows the graphical representation of Rules 4 and 6. Rule 4 relates two tracks (each track joined two platforms) using two junctions. Rule 6 destroys a connection between three platforms by removing a junction. The coordination rules of Figure 4a,b preserve the style specified by Grammar G_1; this can be proven *statically* using the algorithm defined in [FM96]. Therefore these coordination rules preserve the connectivity properties of Grammar G_1. Elimination rule 6, for instance, removes a path between the platform p_3 and platforms p_1 and p_2 but ensures that p_3 remains connected to another part of the network (through e). Note that when a rule changes the number of connections of elements, it indicates the change explicitly using different terminals (*cf.* Rule 3 where $\mathbf{Platform}_1$ is changed into $\mathbf{Platform}_2$).

There are many (possibly quite natural) *incorrect rules* whose application destroys the underlying style. Consider, for instance, the rules of Figure 4c. Rule 7 may obviously disconnect a L-controller by removing the last track connected to it and thus does not preserve the style defined by Grammar G_1. Rule 8 is trickier. Analogous to Rule 4, it connects two tracks

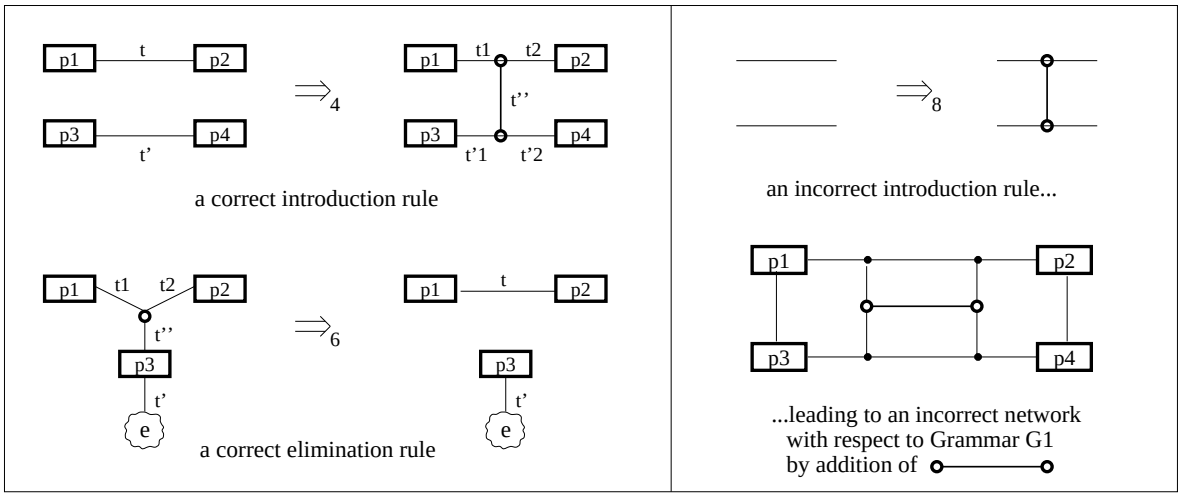


Figure 5: Graphical Representation of Correct and Incorrect Coordination Rules

by introducing two junctions (*cf.* Figure 5 right), however it can be applied in much less restricted contexts and can be used, for instance, to create the network shown in Figure 5 right which does not belong to the style defined by Grammar G_1. The two incorrect rules (7 and 8) are rejected by the verification algorithm.

The correctness of Rules 1–6 with respect to Grammar G_1 guarantees that we meet the requirements set by the ground level of the architecture, namely the connectivity properties. We are now interested in verifying another important property: each L-controller must be controlled by exactly one R-controller. In our design, this requirement depends on the second architectural view, the hierarchy of controllers. So Rules 1–6 must be completed with terminals of Grammar G_2 when they have an impact on the control hierarchy. Consider for instance Rule 1 of Figure 4 – the insertion of a platform on an existing track. We augment this rule with information about controllers of the hierarchy by linking each L-controllers to an R-controller, yielding the Rule $1'$ shown in Figure 6.

This new rule must be checked with respect to Grammars G_1 and G_2. Correctness with respect to the Grammar G_1 still holds because the restriction of Rule $1'$ to the G_1-view yields the initial rule. In order to ensure the correctness with respect to the whole architecture, it is therefore sufficient to check the augmented rule $1'$ against Grammar G_2. With decomposition into architectural views, we can define richer and richer coordination rules, step by step, while ensuring their correctness separately with respect to each view.

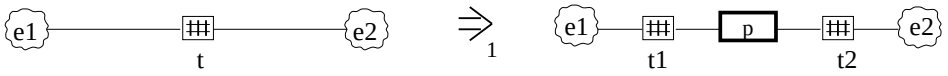
3.3.2 Evolution of the Control Hierarchy

Figure 7 shows a set of rules offering facilities to reorganize the tree of controllers dynamically. An overloaded HL-controller can be split using the first rule which introduces a new HL-controller with at least one sub-controller to supervise. The second rule is useful to transfer sub-controllers from an HL-controller to one of its siblings. We guarantee that the transfer of a sub-controller does not leave an HL-controller without sub-controllers to supervise[4]. The third rule is the opposite of the first one: it destroys a controller transferring its last sub-

[4]This is why Rules 1 and 2 of Figure 7 take two sub-controllers for c'_1.

Initial Rule (Rule 1 of Figure 4 in expanded form)

$\mathbf{Track}(t)$, $\mathbf{connection}\{(t, e_1), (t, e_2)\}$ $\Rightarrow_1$ $\mathbf{Track}(t_1)$, $\mathbf{Platform}_2(p)$, $\mathbf{Track}(t_2)$, $\mathbf{connection}\{(t_1, e_1), (t_1, p), (p_2, p), (t_2, e_2)\}$



Augmented Rule

$[\mathbf{R\text{-}Ctrler}(c), \mathbf{ctrl}\{(c, e_1), (c, e_2)\}]$:
$\mathbf{ctrl}(c, t)$, $2\times \mathbf{R\text{-}freelink}(c)$, $\mathbf{Track}(t)$, $\mathbf{connection}\{(t, e_1), (t, e_2)\}$ $\Rightarrow_{1'}$ $\mathbf{ctrl}\{(c, t_1), (c, p), (c, t_2)\}$, $\mathbf{Track}(t_1)$, $\mathbf{Platform}_2(p)$, $\mathbf{Track}(t_2)$, $\mathbf{connection}\{(t_1, e_1), (t_1, p), (t_2, p), (t_2, e_2)\}$

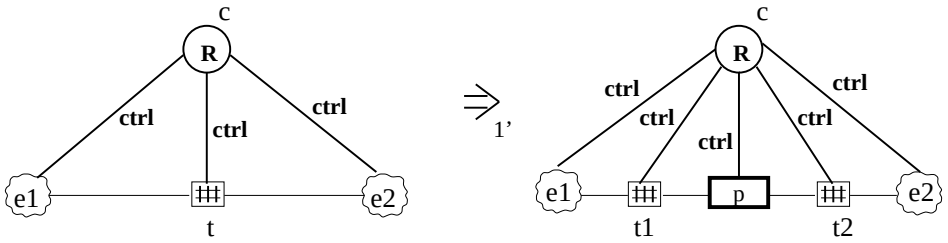


Figure 6: Example of a coordination rule and its augmented version

Split: creation of a new HL-controller with at least one sub-controller to supervise:

$$\left[\mathbf{HL\text{-}Ctrler}(c),\ \mathbf{supervise}(c, c'_1),\ \mathbf{HL\text{-}Ctrler}(c'_1),\ \mathbf{supervise}(c'_1, c''_1)\right]:$$
$$\mathbf{HL\text{-}freelink}(c),\ \mathbf{supervise}(c'_1, c''_2) \quad \Rightarrow_1 \quad \mathbf{supervise}(c, c'_2),\ \mathbf{HL\text{-}Ctrler}(c'_2),\ \mathbf{supervise}(c'_2, c''_2),\ \mathbf{HL\text{-}freelink}(c'_1),\ (max-1)\times \mathbf{HL\text{-}freelink}(c'_2)$$

Transfer of a sub-controller from a HL-controller to a child:

$$\left[\mathbf{HL\text{-}Ctrler}\{c, c'_1, c'_2\},\ \mathbf{supervise}\{(c, c'_1), (c, c'_2), (c'_1, c''_1)\}\right]:$$
$$\mathbf{supervise}(c'_1, c''_2),\ \mathbf{HL\text{-}freelink}(c'_2) \quad \Rightarrow_2 \quad \mathbf{HL\text{-}freelink}(c'_1),\ \mathbf{supervise}(c'_2, c''_2)$$

Merge: destruction of a HL-controller after transfer of its last sub-controller to a sibling:

$$\left[\mathbf{HL\text{-}Ctrler}\{c'_1, c'_2\},\ \mathbf{supervise}\{(c, c'_1), (c, c'_2)\}\right]:$$
$$\mathbf{HL\text{-}freelink}(c'_1),\ \mathbf{supervise}(c'_2, c''),\ \mathbf{HL\text{-}Ctrler}(c'_2),\ (max-1)\times \mathbf{HL\text{-}freelink}(c'_2) \quad \Rightarrow_3 \quad \mathbf{supervise}(c'_1, c'')$$

Lifting:

$$\left[\mathbf{HL\text{-}Ctrler}\{c, c'\},\ \mathbf{supervise}\{(c, c'), (c', c''_1)\}\right]:$$
$$\mathbf{HL\text{-}freelink}(c),\ \mathbf{supervise}(c', c''_2) \quad \Rightarrow_4 \quad \mathbf{supervise}(c, c''_2),\ \mathbf{HL\text{-}freelink}(c')$$

Deepening:

$$\left[\mathbf{HL\text{-}Ctrler}\{c, c'_1\},\ \mathbf{supervise}(c, c'_1)\right]:$$
$$\mathbf{supervise}(c, c'_2),\ \mathbf{HL\text{-}freelink}(c'_1) \quad \Rightarrow_5 \quad \mathbf{HL\text{-}freelink}(c),\ \mathbf{supervise}(c'_1, c'_2)$$

Figure 7: Coordination Rules for the Hierarchy of Controllers

controller to another HL-controller. The last two rules are for moving an HL-controller up or down. Note that these rules only need to be verified for the style defined by Grammar G_2 because the restriction to the terminals of Grammar G_1 yields the rule $\emptyset \Rightarrow \emptyset$ which is obviously correct.

4 Implementation in ConCoord

The formal specification of the railway control system developed in the previous section serves as a basis to derive an implementation of the control system. In this section, we implement the first coordination rule of Figure 4 in its augmented version which is denoted Rule $1'$ in Figure 6 and models the addition of a platform to an existing track in a railway region. In order to remain at the same abstraction level as the graph grammars, we use ConCoord's *coordination language*: CCL [Hol96]. This allows us to express the modifications of the control network and hierarchy described in Rule $1'$ referring solely to the interfaces of the involved local and regional controllers (these interfaces appear in the next section). The implementations of

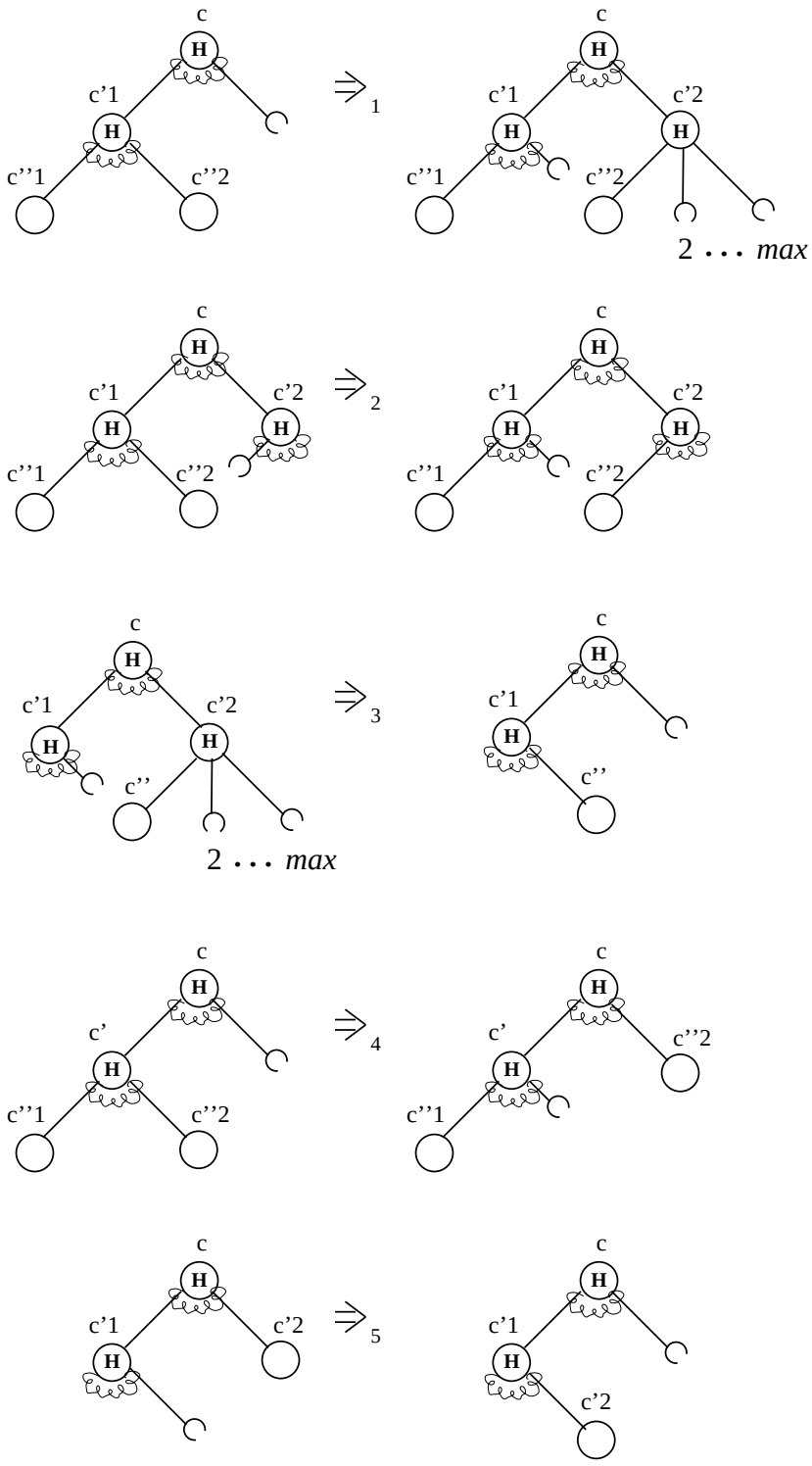


Figure 8: Graphical Representation of Coordination Rules for the Hierarchy of Controllers

Each controller implementation can be written in a different *computation language* which is a sequential language with a few extensions for communication.

4.1 The Interfaces of Local and Regional Controllers

```
component<t_device> L_Ctrler(int id,int n_neigh,L_init<t_device> init)
{ inoutport<t_neigh> neigh[n_neigh];
  inoutport<t_reg> reg;
  states no_train (int[n_neigh]);
}

component R_Ctrler(int id,int max_local,R_init init)
{ inoutport<t_reg> local[max_local];
  inoutport<t_high> higher;
  states add_platform (int,int,L_init<track>,int,L_init<platform>,
                       int,L_init<track>);
}
```

In ConCoord a system component is an instance of a type called *component* whose interface may declare generic and initialisation parameters, ports and states. One gives values to generic and initialisation parameters when instantiating a component. Above, we declare a generic parameter `t_device` in `L_Ctrler` which allows us to parameterise the behaviour of instances of `L_Ctrler` by the type of device to be controlled (*i.e.* track, platform or junction). For instance, we denote a local track controller by `L_Ctrler<track>`. A local controller is initialised with a unique identifier `id`, the number of its neighbouring devices (`n_neigh`) (*e.g.* two for a track controller) and state information `init` (*e.g.* a track length for a track controller). A regional controller, *i.e.* an instance of `R_Ctrler`, is initialised with a unique identifier `id`, the maximum number of local controllers forming its region (`max_local`) and state information `init` (*e.g.* starting topology of the region). Parameters related to component interactions like the initialisation parameters `n_neigh` and `max_local` are derived from the graph grammars; others like the generic parameter `t_device` are proper to the implementation.

An instance of a *component* interacts with other system components via its ports *inoutport* by sending/receiving messages whose type is defined between angle brackets in the code shown above. A local controller for a railway device interacts with the local controllers for its neighbouring devices by the port array `neigh` (one element per neighbouring controller) and with its region controller via the port `reg`. A regional controller interacts with its region local controllers via the port array `local` (one element per potential local controller) and with its higher-level controller by means of the port `higher`. Though ConCoord does not actually provide a port type *inoutport*, we use it here to remain at the same level of abstraction as the graph grammars. In a ConCoord implementation, we would build each *inoutport* defined above in terms of various input or output ports. For example, instead of the port `reg` in `L_Ctrler` we would declare four ports: `req_state` for a local controller to be requested its state by its regional controller, `give_state` for a local controller to reply to such a request, `alarm` for a local controller to notify an alarm to its regional controller and `reg_cmd` for a local controller to get a command from its regional controller. This is shown in the code below.

```
inport<int> req_state;
outport<t_local_state> give_state;
outport<t_local_alarm> alarm;
inport<t_local_cmd> reg_cmd;
```

In the previous section, we have defined architectural modifications of the railway control system by means of coordination rules whose *lhs* refers to the system architecture but not to its component states. In practice, the execution of such rules is triggered by both the system structure and its component states. In the declaration *states* of `L_Ctrler` and `R_Ctrler`, we define execution states which are relevant in the triggering of the addition of a platform to a railway region. It seems reasonable to require that no trains are currently on the devices affected by this modification; this information is provided by a state variable `no_train` in `L_Ctrler` which indicates the existence/absence of trains moving from a device towards its neighbouring devices. As already said, the monitoring of the railway control system and thus its interfacing with users occurs through the hierarchy controllers. In particular, a user requests to a regional controller the addition of a platform to its region; this information is represented in `R_Ctrler` by a state variable `add_platform` whose parameters provide sufficient information to execute the platform addition as we detail below. A system designer must foresee the component states associated to dynamic architectural modifications when specifying coordination rules. Such states can be included in the *lhs* of coordination rules but they are meaningless for the verification algorithm.

4.2 The Addition of a Platform

In Rule 1′ of Figure 6 the addition of a platform to an existing track in a railway region is modelled as the replacement of a track controller `t` by a track controller `t1`, a platform controller `p` and a track controller `t2`. The scope of the modification is a single region supervised by a regional controller `c`. The track controller `t` is neighbouring two local controllers named `e1` and `e2`. As `e1` and `e2` can manage trains at either a platform or a junction, four CCL conditional statements are necessary to express this rule. In the statement *when* below `e1` and `e2` are two platform controllers. This statement defines a condition on the controller states and the system structure which triggers the execution of architectural actions.

```
when (c,t,e1,e2,e1_id,e2_id):(R_ctrler c;L_Ctrler<track> t;
                              L_Ctrler<platform> e1,e2; int e1_id, e2_id
          // Condition on the state of system components
  c.add_platform(t_id,t1_id,t1_init,p_id,p_init,t2_id,t2_init)
  and e1.no_train(e1_nt) with (e1_nt[1]==0)
  and t.no_train(t_nt) with (t_nt[0]==0 and t_nt[1]==0)
  and e2.no_train(e2_nt) with (e2_nt[0]==0)
          // Condition on the system architecture
  and c.local[t_id]--t.reg and c.local[e1_id]--e1.reg
  and c.local[e2_id]--e2.reg
  and e1.neigh[1]--t.neigh[0] and t.neigh[1]--e2.neigh[0])
          // Architectural Modification
    => forall (c,t,e1,e2,e1_id,e2_id)
      { kill t;
        create    track t1(t1_id,2,t1_init), platform p(p_id,2,p_init)
                  track t2(t2_id,2,t2_init),
        bind      e1.neigh[1]--t1.neigh[0],t1.neigh[1]--p.neigh[0],
                  p.neigh[1]--t2.neigh[0],t2.neigh[1]--e2.neigh[0],
                  c.local[t1_id]--t1.reg,c.local[p_id]--p.reg,
                  c.local[t2_id]--t2.reg;
      };
```

The first part of the condition queries on the state `add_platform` of the regional controller `c` which indicates the existence of a user request for the addition of a platform in the region. Its parameters provide the identification of the track controller `t_id` onto which the platform has to be placed and values for the initialisation parameters of the track and platform controllers to be created (see statement *create*). The state `no_train` of the local controllers `e1`, `t` and `e2` ensures the absence of trains moving from (to) the platform controlled by `e1` via the track controlled by `t` to (from) the platform controlled by `e2`. We use the indices `0` and `1` to refer to the left and right neighbouring sides. The second part of the condition queries on the system architecture defined in the *lhs* of Rule $1'$. In ConCoord, a channel between two ports is represented by the binding symbol '$--$' and is mandatory for communication to happen via these ports. The regional controller `c` manages the local controllers `t`, `e1` and `e2` (see above '`local[]--reg`' representing **ctrl** of Grammar G_2). The reader should notice that `t` is identified by `t_id` from `add_platform`. The track controlled by `t` is neighbouring the platforms controlled by `e1` and `e2` (see above '`neigh--neigh`' representing **connection** of Grammar G_1). The second part of the statement *when* (following '=>') details the architectural modifications: the removal of the track controller `t`, the creation of the controllers `t1`, `p` and `t2` (initialised using the parameters of `add_platform`) and the binding of their ports as defined by the *rhs* of the coordination rule. In the statement *when* above, we have directly transcribed the *lhs* and *rhs* of coordination Rule $1'$ into architectural conditions and actions. This step can be fully automated. In a second step we have refined the coordination rule introducing the notion of controller states.

5 Discussion of our approach

In this paper, we have discussed three successive phases of the development of a railway control system: an informal design, a formal definition of its architecture style using graph grammars and the dynamic evolution of the architecture, and a first stage of implementation in terms of ConCoord.

From a software engineering viewpoint, the formal framework we have presented has several advantages. It supports the static verification of some structural properties of a system, those which rely on the architecture style. It is sufficiently powerful to permit the definition of complex systems in terms of various complementary architectural views. Similar to the $4+1$ view model of architectures [Kru95], views can be used to represent (partial) aspects of an architecture, but unlike the $4+1$ model our views are completely formalized. Moreover, the architecture of a railway control system presented here is fully scalable and extensible in the sense required in [dJ97]: the architecture style defined using the two graph grammars captures arbitrarily complex architectures and the coordination rules enable the dynamic extension of an architecture. Finally, the formal definition of a system architecture and its dynamic evolution provide a strong basis for the derivation of an implementation of the system.

This case study has revealed some shortcomings of our framework that require future work to facilitate its use. First, establishing the relationship between a set of possible architectures and its description as a graph grammar is not always intuitive. We believe that it is possible to develop tools for producing instances of minimal size of an architecture style from a graph grammar. This would be helpful to grasp the architectures that a grammar can produce. Second, we had to prove by induction on the production rules the properties of Grammar

G1 and G2 that we announce in subsection 3.2. Since these properties are expressible and decidable in monadic second order logic [Cou90], we would like to verify them automatically; however no practical tools exists.

One of the most attractive features of the formal framework we use is the static verification of coordination rules with respect to graph grammars. This feature requires the graph grammars to be context-free. This restriction limits their expressiveness. For example, it is not possible to describe a hierarchy with an unbounded number of siblings and arbitrary interactions between them. With such a structure, we could represent a control hierarchy in which siblings R-controllers collaborate in the same manner that L-controllers of the control network do, without the participation of HL-controllers.Such a control hierarchy could be described using context-sensitive instead of context-free grammars, but would require an adaptation of the verification algorithm in order to ensure its termination for context-sensitive grammars.

In the formal framework, the coordination rules express local changes and cannot deal with global conditions on the system structure, such as the existence of a path between to given platforms. For example, consider an elimination rule that removes a track. We would like to impose this rule to be applicable only if it does not remove the only track between two subnets of the railway topology. However, this condition cannot be tested when dealing only with local information since it is equivalent to ask for the existence of another path between the two subnets. This notion of locality is a prerequisite for the automatic verification algorithm, facilitates the understanding of coordination rules and enables the simultaneous application of rules in disjoint parts of the system.

References

[AAG95] G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(4):319–364, October 1995.

[Cou90] B. Courcelle. *Graph rewriting: an algebraic and logic approach*, chapter 5 in: Handbook of Theoretical Computer Science. Elsevier, 1990.

[dJ97] E. de Jong. Software architecture for large control systems: a case study. In *Proc. of the Second Conference on Coordination Models, Languages and Applications*, to appear in LNCS. Springer Verlag, 1997.

[FM96] P. Fradet and D. Le Métayer. Structured gamma. TR 989, IRISA, Rennes, 1996.

[Gar95] D. Garlan. Research Directions in Software Architecture. *ACM Computing Surveys*, 27:257–261, June 1995.

[Hol96] A. A. Holzbacher. A Software Environment for Concurrent Coordinated Programming. In *Proc. of the First Conference on Coordination Models, Languages and Applications*, LNCS 1061, pages 249–266. Springer-Verlag, 1996.

[HPS97] A. A. Holzbacher, M. Périn, and M. Südholt. Modeling railway control systems using graph grammars: a case study. In *Proc. of the Second Conference on Coordination Models, Languages and Applications*, to appear in LNCS. Springer Verlag, 1997.

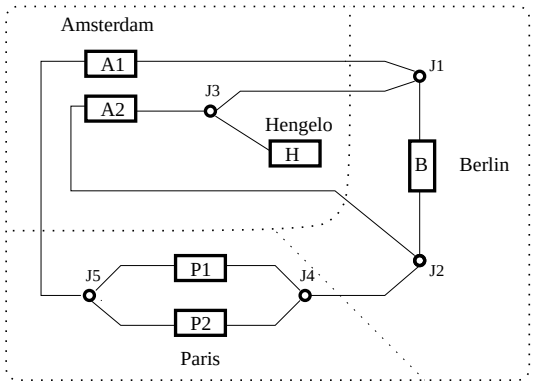
[Kru95] P. B. Kruchten. The $4+1$ view model of architecture. *IEEE Software*, pages 42–50, November 1995.

[Mét96] D. Le Métayer. Software architecture styles as graph grammars. In *In Proc. of the ACM SIGSOFT Symposium of the foundations of Software Engineering*, pages p.15–23, 1996.

[SG95] M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. In *Computer Science Today, Recent Trends and Developments*, LNCS 1000, pages 307–323. Springer-Verlag, 1995.

A Examples of derivation

Case Study Network of Figure 1.

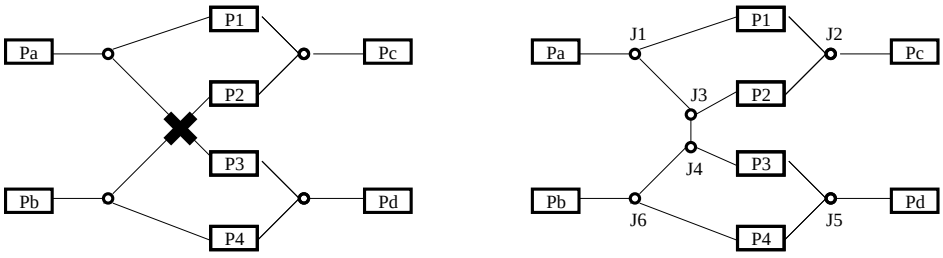


used non-terminals	rules	new non-terminals	new terminals
$RailwayNet$	$\rightarrow_2$	$\underline{SubNet}$	
$SubNet$	$\rightarrow_4$	$\underline{Path(A_1, A_1)}$,	$\mathbf{Platform}_2(A_1)$
$Path(A_1, A_1)$	$\rightarrow_8$	$Path\{(A_1, J_1), \underline{(J_1, J_2)}, \underline{(J_1, J_2)}, \underline{(J_2, A_1)}\}$,	$\mathbf{Junction}\{J_1, J_2\}$
$Path(J_1, J_2)$	$\rightarrow_6$	$Path\{(J_1, B), (B, J_2)\}$,	$\mathbf{Platform}_2(B)$
$Path(J_1, J_2)$	$\rightarrow_6$	$Path\{\underline{(J_1, A_2)}, (A_2, J_2)\}$,	$\mathbf{Platform}_2(A_2)$
$Path(J_1, A_2)$	$\rightarrow_7$	$Path\{(J_1, J_3), (J_3, A_2), (J_3, H)\}$,	$\mathbf{Junction}(J_3)$, $\mathbf{Platform}_1(H)$
$Path(J_2, A_1)$	$\rightarrow_8$	$Path\{(J_2, J_4), \underline{(J_4, J_5)}, \underline{(J_4, J_5)}, (J_5, A_1)\}$,	$\mathbf{Junction}\{J_4, J_5\}$
$Path(J_4, J_5)$	$\rightarrow_6$	$Path\{(J_4, P_1), (P_1, J_5)\}$,	$\mathbf{Platform}_2(P_1)$
$Path(J_4, J_5)$	$\rightarrow_6$	$Path\{(J_4, P_2), (P_2, J_5)\}$,	$\mathbf{Platform}_2(P_2)$

We apply Rule 5 as many time as needed to transform all the remaining paths† into tracks and their connections.

† (A_1, J_1), (J_1, B), (B, J_2), (A_2, J_2), (J_1, J_3), (J_3, A_2), (J_3, H), (J_2, J_4), (J_5, A_1), (J_4, P_1), (P_1, J_5), (J_4, P_2), (P_2, J_5).

Derivation of an Interchanger Network.



a) "real-world" interchanger b) model

We can define an interchanger network using a four way junction as shown above. The derivation presented below proves that the four way junction belongs to the style defined by the railway topology grammar shown in Figure 2.

used non-terminals	rules	new non-terminals	new terminals
$\underline{RailwayNet}$	$\rightarrow_2$	$\underline{SubNet}$	
$\underline{SubNet}$	$\rightarrow_3$	$\underline{Path(P_a, P_c)}$,	$\mathbf{Platform}_2\{P_a, P_c\}$
$Path(P_a, P_c)$	$\rightarrow_8$	$Path\{(P_a, J_1), \underline{(J_1, J_2)}, \underline{(J_1, J_2)}, (J_2, P_c)\}$,	$\mathbf{Junction}\{J_1, J_2\}$
$Path(J_1, J_2)$	$\rightarrow_6$	$Path\{(J_1, P_1), (P_1, J_2)\}$,	$\mathbf{Platform}_2(P_1)$
$Path(J_1, J_2)$	$\rightarrow_7$	$Path\{(J_1, J_3), \underline{(J_3, J_2)}, \underline{(J_3, P_d)}\}$,	$\mathbf{Junction}(J_3)$, $\mathbf{Platform}_1(P_d)$
$Path(J_3, J_2)$	$\rightarrow_6$	$Path\{(J_3, P_2), (P_2, J_2)\}$,	$\mathbf{Platform}_2(P_2)$
$Path(J_3, P_d)$	$\rightarrow_8$	$Path\{(J_3, J_4), \underline{(J_4, J_5)}, \underline{(J_4, J_5)}, (J_5, P_d)\}$,	$\mathbf{Junction}\{J_4, J_5\}$
$Path(J_4, J_5)$	$\rightarrow_6$	$Path\{(J_4, P_3), (P_3, J_5)\}$,	$\mathbf{Platform}_2(P_3)$
$Path(J_4, J_5)$	$\rightarrow_7$	$Path\{(J_4, J_6), \underline{(J_6, J_5)}, (J_6, P_b)\}$,	$\mathbf{Junction}(J_6)$, $\mathbf{Platform}_1(P_b)$
$Path(J_6, J_5)$	$\rightarrow_6$	$Path\{(J_6, P_4), (P_4, J_5)\}$,	$\mathbf{Platform}_2(P_4)$

We apply Rule 5 as many time as needed to transform all the remaining paths[‡] into tracks and their connections.

[‡] (P_a, J_1), (J_2, P_c), (J_1, P_1), (P_1, J_2), (J_1, J_3), (J_3, P_2), (P_2, J_2), (J_3, J_4), (J_5, P_d), (J_4, P_3), (P_3, J_5), (J_4, J_6), (J_6, P_b), (J_6, P_4), (P_4, J_5).

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399